42392.P9771 *PATENT*

UNITED STATES PATENT APPLICATION

FOR

A VIRTUAL TRANSLATION LOOKASIDE BUFFER

INVENTORS:

GILBERT NEIGER
STEPHEN CHOU
ERIK COTA-ROBLES
STALINSELVARAJ JEYASINGH
ALAIN KAGI
MICHAEL A. KOZUCH
RICHARD UHLIG
SEBASTIAN SCHOENBERG

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026

(408) 720-8300

Attorney's Docket No. 42390P9771

"Express Mail" mailing label number EL627464271US

Date of Deposit December 27, 2000
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Assistant Commissioner of Patents and Trademarks, Washington, D.C. 20231.

Michelle Begay
(Typed or printed name of person mailing paper or fee)

Michelle Begay
(Signature of person mailing paper or fee)

A VIRTUAL TRANSLATION LOOKASIDE BUFFER

Field of the Invention

The present invention relates generally to virtual machines, and more specifically to supporting address translation in a virtual machine environment.

Background of the Invention

A conventional virtual-machine monitor (VM monitor) typically runs on a computer and presents to other software the abstraction of one or more virtual machines. Each virtual machine may function as a self-contained platform, running its own “guest operating system” (i.e., an operating system hosted by the VM monitor). The guest operating system expects to operate as if it were running on a dedicated computer rather than a virtual machine. That is, the guest operating system expects to control various computer operations and have an unlimited access to the computer’s physical memory and memory-mapped I/O devices during these operations. For instance, the guest operating system expects to maintain control over address-translation operations and have the ability to allocate physical memory, provide protection from and between guest applications, use a variety of paging techniques, etc. However, in a virtual-machine environment, the VM monitor should be able to have ultimate control over the computer’s resources to provide protection from and between virtual machines.

Thus, an address-translation mechanism is needed that will support attempts of a guest operating system to control address translation while enabling a VM monitor to retain ultimate control over address translation and computer resources.

Brief Description of the Drawings

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

Figure 1 illustrates one embodiment of a virtual-machine environment;

Figure 2 illustrates a prior-art embodiment of an address-translation mechanism that supports a hardware-managed TLB;

Figure 3 is a block diagram of an address-translation system according to one embodiment of the present invention;

Figure 4 is a flow diagram of a method for supporting address translation, according to one embodiment of the present invention;

Figure 5 is a flow diagram of a method 500 for handling events initiated by the guest OS, according to one embodiment of the present invention;

Figure 6 illustrates operation of a virtual TLB that supports IA-32 address translation, according to one embodiment of the present invention;

Figures 7A-7D are flow diagrams of a method for responding to a page fault, according to one embodiment of the present invention;

Figure 8 is flow diagram of a method for responding to an INVPLG instruction issued by a guest OS, according to one embodiment of the present invention;

Figure 9 is a flow diagram of a method for handling an attempt of a guest OS to modify the write-protect bit in control register CR0, according to one embodiment of the present invention;

Figures 10A and B are flow diagram of a method for responding to an operation that may require modification of a virtual TLB using an eager-filling technique, according to one embodiment of the present invention; and

Figure 11 is a block diagram of one embodiment of a processing system.

Description of Embodiments

A method and apparatus for supporting address translation are described. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention can be practiced without these specific details.

Some portions of the detailed descriptions that follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, may refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer-system memories or registers or other such information storage, transmission or display devices.

The present invention also relates to apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may comprise a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable storage medium, such as, but is not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs), EPROMs, EEPROMs, magnetic or optical cards, or any type of media suitable for storing electronic instructions, and each coupled to a computer system bus. Instructions are executable using one or more processing devices (e.g., processors, central processing units, etc.).

The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general-purpose machines may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

In the following detailed description of the embodiments, reference is made to the accompanying drawings that show, by way of illustration, specific embodiments in which the invention may be practiced. In the drawings, like numerals describe substantially similar components throughout the several views. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention. Other embodiments may be utilized and structural, logical, and electrical changes may be made without departing from the scope of the present invention. Moreover, it is to be understood that the various embodiments of the invention, although different, are not necessarily mutually exclusive. For example, a particular feature, structure, or characteristic described in one embodiment may be included within other embodiments. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present invention is defined only by the appended claims, along with the full scope of equivalents to which such claims are entitled.

The method and apparatus of the present invention provide a mechanism for supporting address translation in a virtual machine environment. **Figure 1** illustrates one embodiment of a virtual-machine environment 100, which employs a virtual-machine monitor (VMM) 112. In this embodiment, bare platform hardware 114 comprises a computing platform, which may be capable, for example, of executing a standard operating system (OS) or a virtual-machine monitor (VMM), such as VMM 112. A VMM, though typically implemented in software, may export a bare machine interface, such as an emulation, to higher level software. Such higher level software may comprise a standard or real-time OS, although the invention is not limited in scope in this respect and, alternatively, for example, a VMM may be run within, or on top of, another VMM. VMMs and their typical features and functionality are well-known by those skilled in the art and may be implemented, for example, in software, firmware or by a combination of various techniques.

As described above, a VMM presents to other software (i.e., "guest" software) the abstraction of one or more virtual machines (VMs). **Figure 1** shows two VMs, 102 and 114. Each VM includes a guest OS such as guest OS 104 or 106 and various guest software applications 108-110. Each of guest OSs 104 and 106 expects to control access to physical resources (e.g., memory and memory-mapped I/O devices) within the hardware platform on which the guest OS 104 or 106 is running and to perform other functions. For instance, during address-translation operations, the guest OS expects to allocate physical memory, provide protection from and between software applications (e.g., applications 108 or 110), use a variety of paging techniques, etc. However, in a virtual-machine environment, VMM 112 should be able to have ultimate control over the physical resources to provide protection from and between VMs 102 and 114. This conflict between the expectations of the guest OS and the role of the VMM becomes an issue during address-translation operations initiated by the VM.

A conventional address-translation mechanism is typically based on a translation lookasdide buffer (TLB), an in-processor structure that acts as a cache for previously processed address translations. A TLB may be either hardware-managed (i.e., fully managed by the processor) or software-managed (i.e., accessed by the processors but managed by software). The VMM can manage a software-managed TLB, thereby retaining control over address translation. The problem arises if the processor supports only hardware-managed TLBs over which the VMM has no direct control. The present invention addresses this problem by providing an address-translation mechanism that supports attempts of the guest OS to control address translation while enabling the VMM to retain ultimate control over address translation and computer resources.

A prior art address-translation mechanism that uses a hardware-managed TLB will now be described in more detail. Although address translation features are described below as applied to IA-32 microprocessors, many of these features apply as well to a variety of other microprocessors that support hardware-managed TLBs.

Referring to **Figure 2**, address translation is controlled by a TLB 202 and a page-table hierarchy 204. Page-table hierarchy 204, which is referenced by the processor's control register CR3 (i.e., block 206), is a translation data structure used to translate a virtual memory address into a physical memory address when paging is enabled. Page-table hierarchy 204 includes a page directory (PD) 208, a set of page tables (PTs) 210, and multiple page frames (Fs) 212. Page-table hierarchies and their typical features and functionality are well known by those skilled in the art.

Typically, translation of a virtual memory address into a physical memory address begins with searching TLB 202 using either the upper 20 bits (for a 4 KB page) or the upper 10 bits (for a 4MB page) of the virtual address. If a match is found (a TLB hit), the upper bits of a physical page frame that are contained in TLB 202 are conjoined with the lower bits of the virtual address to form a physical address. If no match is found (a TLB miss), the processor consults the page table hierarchy 204 to determine the virtual-to-physical translation, which is then cached in TLB 202.

Each entry in PD 208 and PTs 210 typically includes three bits that control use of translations generated by this entry: the present (P) flag, the user/supervisor (U/S) flag, and the read/write (R/W) flag. The P flag indicates whether or not the structure referenced by the entry is valid. If the translation process accesses a PD entry or a PT entry whose P bit is clear, the process stops at this point and a page fault is generated. The U/S flag controls access based on privilege level. The R/W flag controls access based on access type (i.e., read or write).

In addition, each entry in PD 208 and PTs 210 contains two bits that are automatically set by the processor on certain accesses. These bits are an accessed (A) bit and a dirty (D) bit. The A bit of a PD entry or PT entry that points to a page frame is set whenever the page frame is read or written; the D bit is set whenever the page frame is written. The A bit of a PD entry that points to a PT is set whenever that PT is accessed using page table hierarchy 204.

The translations cached in TLB 202 include information about page-access rights (i.e., information derived from the U/S and R/W bits) and page usage (the A and D bits). If the page-table hierarchy is modified, TLB 202 may become inconsistent with the page-table hierarchy 204 if a corresponding address translation exits in TLB 202. Typically, the processor allows software to resolve such an inconsistency. For instance, IA-32 processors allow software to invalidate cached translations in TLB 202 by using the INVLPG instruction, which takes a virtual address as an operand. Any translation for that virtual address is removed from TLB 202. In addition, when the address space (i.e., the virtual-to-physical mapping) is changed completely, numerous translations may need to be removed from TLB 202. This may be done by loading CR3 (which contains the base address of the page directory), thereby removing all translations from TLB 206. CR3 may be loaded using a MOV instruction or a task switch.

The instructions that explicitly manipulate TLB 202 can be performed only by the privileged software. For instance, for IA-32 microprocessors, INVPLG instructions and MOV CR instructions can only be performed by software running at the most privileged level (i.e., level 0), and the guest OS may require that a task switch be performed only by the most privileged software. As described above, in the virtual-machine environment, the VMM should be able to have ultimate control over physical resources including TLB 202 and to limit access to these resources by guest OSs. In some computer architectures (e.g., architectures using IA-32 microprocessors), this may be accomplished using a guest-deprivileging technique.

Guest deprivileging forces all guest software to run at a hardware privilege level that does not allow that software access to certain hardware resources. For instance, for IA-32 microprocessors, the nature of page-based protection is such that all guest software runs at the least privileged level (i.e., privilege level 3). In the case of some microprocessors (e.g., IA-32 microprocessors), guest deprivileging causes a trap when guest software attempts to access such hardware resources as TLB 202 (e.g., when the guest OS issues any of the instructions 216). The traps can be handled by the VMM, thereby allowing the VMM to retain ultimate control over physical resources.

Guest deprivileging, however, may cause a ring compression problem. That is, because all guest software may run at the same privilege level, the guest operating system may not be protected from guest software applications. One embodiment of the present invention addresses this problem by maintaining different translation data structures (e.g., page-table hierarchies) for guest software at different privilege levels as will be described in greater detail below.

The present invention provides an address-translation mechanism that supports virtualization. **Figure 3** is a block diagram of an address-translation system 300, according to one embodiment of the present invention. System 300 includes a guest translation data structure 308 and a virtual TLB 302. The guest translation structure 308 indicates how the guest OS intends to translate virtual memory addresses to physical memory addresses. One example of such a translation data structure is a page-table hierarchy 104 described above in conjunction with **Figure 2**. However, various other translation data structures may be used with the present invention without loss of generality. The guest translation data structure 308 is managed by the guest OS, which can access and modify any entry in the guest translation data structure.

The virtual TLB 302 supports the guest OS's attempts to control address translation by responding to address-translation operations performed by the guest OS with an interface that emulates the functionality of the processor's physical TLB. Thus, the guest OS is forced to believe that it deals with the physical TLB.

The virtual TLB 302 includes a physical TLB 304 and an active translation data structure 306. The active translation data structure 306 derives its format and content from the guest translation data structure 308. The active translation data structure 306 is created and managed by the VMM. The VMM resolves inconsistencies between the guest translation data structure 308 and the active translation data structure 306 using techniques analogous to those employed by the processor in managing the TLB.

The physical TLB 304 is loaded by the processor with address translations derived from the active translation data structure 306. Accordingly, address translation is controlled by the processor, which manages the physical TLB 304, and by the VMM, which manages the active translation data structure 306. Thus, the virtual TLB provides a mechanism for tolerating and supporting the guest OS's attempts to control address translation while allowing the processor and the VMM to retain ultimate control over all address-translation operations.

In one embodiment, which supports guest deprivileging, more than one active translation data structure is used to address ring-compression problems described above. For instance, in a computer architecture using IA-32 microprocessors, one active translation data structure may be maintained for privilege level 3 (the active user translation data structure) and one active translation data structure may be maintained for privilege level 0 (the active supervisor translation data structure).

Figure 4 is flow diagram of a method 400 for supporting address translation, according to one embodiment of the present invention. Method 400 begins with creating a guest translation data structure that will be used by a guest OS for performing address-translation operations (processing block 404). For instance, the guest OS should be able to add, delete or replace entries in the guest translation data structures (e.g., entries in the page directory or page tables), reset flags that control use of the address translations generated by the entries, or otherwise modify the content of the guest translation data structure.

At processing block 406, an active translation data structure is created based on the guest translation data structure. The active translation data structure is managed by the VMM. In one embodiment, a separate active translation data structure is maintained for each virtual machine. Alternatively, two or more virtual machines may share the same active translation data structure. In one embodiment, more than one active translation data structure is maintained for a virtual machine. For instance, in a computer architecture supporting IA-32 processors, an active user translation data structure and an active supervisor translation data structure are maintained for each virtual machine.

At processing block 408, the content of the active translation data structure is periodically modified to conform to the content of the guest translation data structure. The content of the active translation data structure is then used by the processor to cache address translations in the TLB. The combination of the active translation data structure and the TLB is referred to as a virtual TLB because it provides to the guest operating system the functionality analogous to that of the physical TLB, thereby supporting the guest OS's attempts to control address translation. This functionality is provided using several mechanisms, which will be described in greater detail below. Some of these mechanisms force the guest operating system to issue an event which results in passing control of a corresponding address-translation operation to the VMM. The VMM than evaluates the event and performs an appropriate action. **Figure 5** illustrates one embodiment of a method 500 for handling events initiated by the guest OS.

Referring to **Figure 5**, method 500 begins with the VMM's receiving control over an event initiated by the guest OS (processing block 504). In one embodiment, the event initiated by the guest OS may result in a trap that passes control to the VMM. As described above, a trap may be generated as a result of guest deprivileging. However, any other software or hardware technique known in the art may be used to support traps or otherwise enable transfer of control over the event from the guest OS to the VMM. Control may be passed to the VMM in response to various events initiated by the guest OS. Such events may include, for example, events indicating the guest OS's attempts to manipulate the TLB (e.g., for IA-32 microprocessors, these events include instructions 216 shown in **Figure 2**), page faults generated by the processor in response to an operation performed by the guest software, changes of privilege level, and other events that may require the VMM's involvement in order to ensure that VMM remains in control of address translation.

At processing block 506, the event is evaluated. In one embodiment, at decision box 508, decision is made to determine whether the event is caused by an attempt of guest software to change its privilege level. If the determination is positive, a further determination is made as to whether the change in the privilege level is sensitive to page-based protection (decision box 510). If the change is sensitive to page-based protection, then at processing block 512, the control register (e.g., CR3) is reloaded with the physical address of the appropriate active translation data structure (e.g., in the case of IA-32 microprocessors, the appropriate active translation data structure is either the active user data structure for a transition into privilege level 3 or the active supervisor data structure for a transition out of privilege level 3). Otherwise, if the change is not sensitive to the page-based protection, no change to the control register is required.

If the determination made at decision box 508 is that the event initiated by the guest OS is not caused by an attempt of guest software to change its privilege level, a further determination is made at decision box 514 as to whether the event is caused by an explicit attempt of the guest software to modify the TLB. If the determination is positive, then the event was generated due to a possible inconsistency between the virtual TLB and the guest translation data structure. Accordingly, the content of the active translation data structure may be modified to conform to the content of the guest translation data structure (processing block 516). In one embodiment, only the entries in the active translation data structure that are associated with the event are modified. In an alternative embodiment (described in greater detail below in conjunction with **Figures 10A and B**), all entries in the active translation data structure that do not match corresponding entries in the guest translation data structure are modified.

If the determination made at decision box 514 is negative, a further determination is made at decision box 522 as to whether the event is associated with a page fault generated by the processor. If the determination is negative, method 500 ends. Otherwise, a further determination is made as to whether the page fault would occur under normal operation of the guest OS (decision box 524). If this determination is positive, then this event requires action on the part of the guest OS, rather than the VMM. Accordingly, at processing block 518, the VMM passes control over the event back to the guest OS, which will handle the event as intended. If the determination made at decision box 524 is negative, i.e., the page fault would not occur under normal operation of the guest OS, then the content of the active translation data structures needs to be analyzed to evaluate whether it is consistent with the content of the guest translation data structure. If any inconsistency is discovered, the content of the active translation data structures is modified to remove the inconsistency (processing block 516).

It should be noted that, for the sake of simplicity, the description of this embodiment does not focus on events that are indicative of an attempt by the guest software to change the privilege level and modify the virtual TLB at the same time.

Various mechanisms provided by the present invention to support the guest OS's attempts to control address translation will now be described in more detail. As discussed above, these mechanisms are used to provide the guest OS with the functionality analogous to that of the physical TLB. These mechanisms are described below with reference to specific IA-32 features as IA-32 microprocessors support various address translation features that are typical for other microprocessors that support hardware-managed TLBs. However, the scope of the present invention should not be so limited. Instead, the present invention is operable with any processor supporting hardware-managed TLBs. In addition, a wide variety of mechanisms other than those described below may be used with the present invention to provide the functionality analogous to that of the physical TLB without loss of generality.

Figure 6 illustrates operation of a virtual TLB 604 supporting IA-32 address translation, according to one embodiment of the present invention. Virtual TLB 604 includes an active translation data structure represented by an active page-table hierarchy 606 and a physical TLB 608. The active page-table hierarchy 606 derives its entries from a guest translation data structure represented by a guest page-table hierarchy 602. The VMM maintains for each virtual machine the value that the virtual machine expects for the control registers controlling address translation (i.e., CR0, CR2, CR3, and CR4). These control registers are referred to as guest control registers.

As described above, in one embodiment, the VMM creates two active page-table hierarchies 606 (an active user page-table hierarchy and an active supervisor page-table hierarchy) for each virtual machine to ensure that the protection desired by the guest OS is properly emulated. In one embodiment, all entries in both active page-table hierarchies 606 are initially marked invalid (using P flag described above) to emulate the initialization state of the TLB when the TLB has no entries. Subsequently, when guest software presents a virtual address to the processor, the processor finds only invalid entries in the active page-table hierarchy, and a page fault is generated. The page fault transitions control from the guest OS to the VMM. The VMM then copies corresponding entries from the guest page-table hierarchy 602 to the active page-table hierarchy 606. Thus, in this embodiment, the active page-table hierarchy 606 is refilled on page faults. One embodiment of handling page faults will be described in greater detail bellow in conjunction with **Figures 7A-D**.

As described above, the processor sets the accessed (A) bit and dirty (D) bit in the PD entries and PT entries. The virtual TLB emulates this behavior of the processor by maintaining A and D bits in the guest PD and PTs. In one embodiment, when a page is accessed by guest software for the first time, the processor attempts to set the A bit in the corresponding PT entry or PD entry in the active page-table hierarchy 606. In this embodiment, because the entries in the active page-table hierarchy are marked invalid until they are first accessed, the processor's attempt results in a page fault. The VMM monitor then sets the P bit in the in the corresponding PT entry or PD entry in the active page-table hierarchy 606, and sets the A bit in the corresponding PT entries or PD entries in the guest page-table hierarchy 602. The faulting instruction is then re-executed, and it will now not fault because the P bit in the PT entry or PD entry in the active page-table hierarchy has been set. The processor will then set the A bit in the PT entry or PD entry in the active page-table hierarchy.

With respect to the D bit, in one embodiment, the VMM maintains all entries in the active page-table hierarchy 606 as read-only (using the R/W flag) until the D bit is set in the corresponding entries of the guest page-table hierarchy 602. In particular, when guest software attempts to write a page, the processor attempts to set the D bit in the corresponding entry on the active page-table hierarchy 606 that is marked as read-only. As a result, a page fault is generated, and the VMM sets the R/W flag to read/write in the active page-table hierarchy 606 and the D bit in the guest hierarchy 602. The faulting instruction is then re-executed, and it will now not fault because the R/W flag in the PT entry or PD entry in the active page-table hierarchy has been set to read/write. The processor will then set the D bit in the PT entry or PD entry in the active page-table hierarchy.

Guest software is allowed to freely modify the guest page-table hierarchy 602 including changing virtual-to-physical mapping, permissions, etc. Accordingly, the active page-table hierarchy 606 may not be always consistent with the guest page-table hierarchy 602. That is, the active page-table hierarchy 606 may be out-of-date, e.g., it may allow too much access to its entries, provide wrong virtual-to-physical address mapping, etc. However, as described above in conjunction with **Figure 2**, this behavior of the active page-table hierarchy 606 is acceptable (i.e., in a non-virtual machine environment, a page-table hierarchy may become inconsistent with a physical TLB, and problems caused by the inconsistencies are typically resolved using any of the instructions 216). When a problem arises from an inconsistency between the hierarchies 602 and 606, the guest OS, which treats the virtual TLB 604 as a physical TLB, will attempt to change the virtual TLB 604 using one of the instructions 216. These instructions result in the transfer of control from the guest OS to the VMM. The VMM will then determine the cause of the instruction and modify the content of the active page-table hierarchy 606 if necessary. For instance, if the guest page-table hierarchy 602 allows less access than the active page-table hierarchy 606, the VMM is architecturally permitted to allow greater access through the active page-table hierarchy 606 until guest software issues any of the instructions 216 to attempt to remove old entries that became invalid. The use of any of these instructions will transfer control to the VMM, which can then remove the entries referred to by guest software in the issued instruction from the active page-table hierarchy 606.

In one embodiment, the VMM selects the physical-address space that is allocated to guest software. Addresses installed by guest software in the guest CR3 612, in PD entries, and in PT entries (referred as "guest physical addresses") are considered by guest software to be physical addresses. In one embodiment, the VMM may map these addresses to different physical addresses.

As described above, in one embodiment, the VMM maintains more than one active page-table hierarchy (e.g., an active user page-table hierarchy and an active supervisor page-table hierarchy) for each virtual machine. In one embodiment, the U/S flag is set in all entries of the active supervisor page-table hierarchy to allow guest supervisor software to have access to all pages. In one embodiment, the R/W flag is set to 1 (i.e., read and write accesses are allowed) in the entries of the active supervisor page-table hierarchy if the write-protect bit in control register CR0 (CR0.WP) is set to 0. In one embodiment, if CR0.WP is set to 1, the R/W flag should also be set as in the corresponding entries of the guest page-table hierarchy. This requires the VMM to take special action when guest software attempts to modify CR0.WP.

Figures 7A-7D are flow diagrams of one embodiment of a method 700 for responding to a page fault. As described above, a page fault may result from an inconsistency between the active page-table hierarchy and the guest page-table hierarchy. The VMM may then modify the active page-table hierarchy and re-execute the faulting instruction. Alternatively, the hierarchies may already be consistent, and the fault should be handled by the guest OS.

Method 700 begins with evaluating an appropriate page directory entry (PDE) in the active user PD and a corresponding PDE in the active supervisor PD (processing block 704). In one embodiment, these active PDEs are located using the upper 10 bits of the faulting address and the two CR3 values maintained for this virtual machine.

At processing block 706, the intended privilege level of the guest software that generated the fault is determined to identify the corresponding active page-table hierarchy (i.e., either active user hierarchy or active supervisor hierarchy).

At decision block 708, a determination is made as to whether the active PDE being examined caused the page fault. In one embodiment, the determination depends on whether the active PDE is marked not present or its R/W bit and U/S bits are inconsistent with the attempted guest access.

If the determination is positive, the corresponding guest PDE is located (e.g., by using the upper 10 bits of the faulting address and the physical addresses that corresponds to the guest address in the guest CR3), and a decision is made as to whether the guest PDE could also cause the fault (decision box 710), e.g., whether the guest PDE is marked not present or present. If the guest PDE could also be the source of the fault, then the VMM raises the page fault to the guest OS (processing block 712). Otherwise, a further determination is made as to whether a physical address contained in the located guest PDE is valid for the virtual machine being supported (decision box 714). If the address is invalid, the VMM raises a machine check to the guest OS (processing block 716).

If the address is valid for this VM, the examination of the active PDE continues. Specifically, at decision box 718, a determination is made as to whether the active PDE is marked not present. If the determination is positive, the active user PDE and the active supervisor PDE are modified to correspond to the guest PDE. In particular, at decision box 730, a determination is made as to whether the guest PDE contains a page base address (i.e., if PS=1). If the guest PDE does not contain the page base address (i.e., it contains a PT base address instead), then two aligned 4 KB active PTs (active user PT and active supervisor PT) are allocated and marked invalid (processing block 732). Next, the page-table base addresses in the active user PDE and the active supervisor PDE are set to the physical addresses of the corresponding allocated PTs. Method 700 then proceeds to processing block 738.

If the determination made at decision box 730 is positive, the page base addresses in the two active PDEs are set to be the physical address that corresponds to the guest address in the guest PDE (processing block 736). Next, at processing block 738, the P and PS flags in the active PDEs are set to match the values of these flags in the guest PDE. At processing block 740, the U/S flag in the active supervisor PDE is set to 1 and the U/S flag in the active user PDE is set to the value of this flag in the guest PDE. At processing block 744, the A bit is set to 1 in the guest PDE.

Further, a determination is made as to whether the D bit is set to 0 in the guest PDE (decision box 746). If this determination is negative or if PS=0 (decision box 754), then a determination is made as to whether the guest software expects CR0.WP to be set to 0 (decision box 748). If the guest software does expect CR0.WP=0, then the R/W flag in the active supervisor PDE is set to 1 and the R/W flag in the active user PDE is set to the value of this flag in the guest PDE (processing box 752), and method 700 proceeds to processing block 768. Alternatively, if the guest software expects CR0.WP to be set to 1, then the R/W flag in the active supervisor PDE and in the active user PDE is set to the value of this flag in the guest PDE (processing block 750), and method 700 proceeds to processing block 768.

If the determination made in box 746 is positive, i.e., D=0 in the guest PDE, then a further decision is made as to whether the PS flag is set to 0 (decision box 754). If PS=0, method 700 proceeds to decision box 748. Otherwise, if PS=1, then yet further determination is made as to whether the attempted access is a write (decision box 755). If the attempted access is not a write, then the R/W flags in the active supervisor PDE and in the active user PDE are set to 0 (processing block 757), and method 700 proceeds to processing block 768. Alternatively, if the attempted access is indeed a write, then the D bit in the guest PDE is set to 1 (processing block 756) and the R/W flag in the active user PDE is set to this flag's value in the guest PDE (processing block 758). Next, if the guest software expects CR0.WP to be set to 0 (decision box 760), the R/W flag is set to 1 in the active supervisor PDE (processing block 764). Otherwise, if the guest software expects CR0.WP to be set to 1, then the R/W flag in the active supervisor PDE is set to match the value of this flag in the guest PDE (processing block 762).

Afterwards, at processing block 768, the INVLPG instruction is executed with the faulting address, and at processing block 770, the faulting instruction is re-executed.

Returning to decision box 708, if the determination is made that the active PDE is not the source of the page fault, then at decision box 820, a decision is made as to whether this active PDE refers to a 4 MB page, i.e., whether PS=1. If it is determined that the PS flag is set to 1 for this active PDE, it means that the fault resulted from an inconsistency between the active page-table hierarchy and the physical TLB. The VMM then executes the INVLPG instruction (processing block 822) and re-executes the faulting instruction (processing block 824). Alternatively, if the PS flag is set to 0 in the active PDE and the corresponding guest PDE, then the active user PTE and the active supervisor PTE are located (e.g., by using bits 21-12 of the faulting address and the above physical addresses of the active user PDE and the active supervisor PDE) and an appropriate active PTE (i.e., active user PTE or active supervisor PTE) is identified by determining the intended privilege level of the guest software that generated the fault. Next, a determination is made as to whether this active PTE caused the page fault (decision box 826). If this determination is negative, then the fault resulted from an inconsistency between the active page-table hierarchy and the physical TLB. The VMM executes the INVLPG instruction (processing block 828) and re-executes the faulting instruction (processing block 830).

If the determination is positive, i.e., the active PTE is the source of fault, the corresponding guest PTE is located (e.g., by using bits 21-12 of the faulting address and the physical addresses that corresponds to the guest page-table base address in the guest PDE), and a decision is made as to whether the guest PTE could also cause the fault, e.g., whether guest PTE is marked not present (decision box 832). If the guest PTE is the source of fault, then the VMM raises the page fault to the guest OS (processing block 834). Otherwise, method 700 continues with determining whether a physical address contained in the located guest PTE is valid for the virtual machine being supported (decision box 836). If the address is invalid, the VMM raises a machine check to the guest OS (processing block 838).

If the address is valid for this VM, the examination of the active PTE continues. Specifically, at decision box 840, a determination is made as to whether the active PTE as marked not present. If the determination is positive, the active user PTE and the active supervisor PTE are modified to correspond to the guest PTE (processing block 846). In one embodiment, the active PTEs are modified in the manner used for modification of the active PDEs for which PS=1 in the corresponding guest PDEs (described above in conjunction with **Figure 7B**).

Alternatively, if the active PTE is marked present, a determination is made at decision box 842 as to whether certain conditions apply (i.e., whether the attempted access is a write, D=0 in the guest PTE, and the active PTE has caused the page fault solely because its R/W flag is set to 0). If the determination is positive, then at processing block 848, the D flag is set to 1 in the guest PTE. The R/W flag in the guest user PTE is set to the value of the R/W flag in the guest PTE. R/W is set to 1 in the active supervisor PTE to 1 if the guest software expects CR0.WP to be set to 0; otherwise, this flag is set to its value in the guest PTE. If the determination made at box 842 is negative (i.e., the active PTE is marked present and none of the above conditions apply), the page fault is raised to the guest OS (processing block 844).

After completing either of processing blocks 846 or 848, the VMM executes the INVLPG instruction (processing block 850) and re-executes the faulting instruction (processing block 852).

Returning to decision box 718, if the active PDE entry is marked present, a further determination is made at decision box 720 as to whether a set of conditions is satisfied. A first condition within the set of conditions requires that the attempted access be a write. The second condition requires that PS=1 and D=0 in the guest PDE. The third condition requires that the PDE caused the fault solely because its R/W is set to 0. If any of these three conditions is not satisfied, then the page fault is raised to the guest OS (processing block 722). Otherwise, if all of these conditions are satisfied, then the D bit is set to 1 in the guest PDE (processing block 802), and a decision is made as to whether the guest software expects CR0.WP be set to 0 (decision box 804). If the determination is negative, then R/W in the active supervisor PDE is set to match the value of this flag in the guest PDE (processing block 806), and method 700 proceeds to processing block 809. Alternatively, if the guest software expects CR0.WP be set to 0, then R/W is set to 1 in the active supervisor PDE (processing block 808). In either case, R/W in the active user PDE is set to match the value of this flag in the guest PDE (processing block 809). Further, the VMM executes the INVLPG instruction (processing block 810) and re-executes the faulting instruction (processing block 812).

Figure 8 is flow diagram of one embodiment of a method 880 for responding to an INVPLG instruction issued by a guest OS. As described above, typically an OS can use INVLPG to remove entries that are no longer valid from the physical TLB. Since the guest OS considers the active translation data structure to be a part of the physical TLB, it issues the INVLPG instruction to resolve any problem caused by an inconsistency between the active translation data structure and the guest translation data structure. An attempt of the guest OS to execute INVLPG results in transfer of control from the guest OS to the VMM. The VMM then modifies the active translation data structure (e.g., the active user page-table hierarchy and the active supervisor page-table hierarchy) to emulate the desired effect of INVPLG.

Method 880 begins with locating the relevant active PDE (processing block 884). In one embodiment, the active PDE is located using the upper 10 bits of the instruction operand address and the current value of CR3.

At decision box 886, a determination is made as to whether the active PDE refers to a 4 MB (i.e., whether PS=0). If the determination is negative, the active PDE is marked not present (i.e., the P flag is set to 0), and method 880 proceeds to processing block 900. Alternatively, if PS=0 and the active PDE is marked present, then the relevant active PTE is located (processing block 890) and its P flag is set to 0 (processing block 892). In one embodiment, the active PTE is located using bits 21-12 of the operand address and the PT base address in the PDE.

Further, at processing block 894, all entries in the active PT are examined (processing block 894), and determination is made as to whether all these entries are now marked as not present. If the determination is negative, method 880 proceeds to processing block 900. Otherwise, if all PTEs in this active PT are marked not present, then the active PT is deallocated and the P flag in the active PDE is set 0. Afterwards, the VMM executes INVLPG with the faulting address (processing block 900) and control returns to the guest OS.

The guest OS may also attempt to load from or store to CR3 or initiate task switch, causing a change of address space, which may necessitate invalidation of the entire TLB. As described above, any of these attempts will result in transferring control from the guest OS to the VMM. The VMM can then modify the active page-table hierarchy to emulate the desired effect of any of the above operations (i.e., the effect of removing the cached address translations). In one embodiment, the VMM deallocates all active PTs that have been allocated, marks both active PDEs as invalid, reloads CR3 with its current value (to flush the physical TLB), and then returns control to the guest OS.

Figure 9 is a flow diagram of one embodiment of a method 950 for handling an attempt of a guest operating system to modify the write-protect bit in control register CR0. As described above, the content of the guest supervisor page-table hierarchy depends on the value that the guest OS intends to establish for CR0.WP. Specifically, this intended value affects the values of the R/W flags in the guest PDEs and PTEs. Accordingly, the VMM takes control over the guest OS's attempt to modify CR0.WP and performs a set of actions illustrated in **Figure 9**.

Method 950 begins with making a determination as to whether the guest OS attempts to set CR0.WP to 1. If the determination is negative, it means that guest supervisor software may be allowed to write to pages that have been protected. In this case, the VMM does not need to take any additional actions. That is, if the guest supervisor software attempts to write to the protected pages, a page fault will be generated, and the VMM will correct the situation as described above in conjunction with **Figures 7A-7D**.

Alternatively, if the guest OS attempts to set CR0.WP to 1, it may result in protecting some pages from writes by guest supervisor software. The VMM then needs to modify the active supervisor page-table hierarchy accordingly. In one embodiment, the VMM evaluates each active supervisor PDE. Specifically, the VMM starts with the first entry in the active supervisor PD (processing block 955), examines this active supervisor PDE (processing block 956), and determines whether the R/W flag is set to 1 in this active supervisor PDE (decision box 958). If the determination is positive, the R/W flag in the active supervisor PDE is set to the value of this flag in the corresponding guest PDE (processing block 960).

Next, at decision box 962, a determination is made as to whether the active supervisor PDE being examined is marked present and refers to a page table (i.e., PS=0). If the determination is negative, method 950 proceeds to decision box 974. Alternatively, the VMM locates the active supervisor page table addressed by the PDE (processing block 964) and evaluates each active supervisor PTE. Specifically, the VMM begins with the first entry in the active supervisor PT (processing block 965), examines this active supervisor PTE (processing block 966), and determines whether its R/W flag is set to 1 (decision box 968). If the determination is positive, then the R/W flag is set to this flag's value in the guest PTE (processing block 970).

After all active supervisor PTEs are examined, the VMM evaluates the next active supervisor PDE in the same manner. Method 950 ends when no more entries remain in the active supervisor PD.

As described above in conjunction with **Figures 7A-7D and 8**, in some embodiments, the VMM evaluates only the active PDEs and PTEs that correspond to the virtual address that generated a page fault or that was the operand of an INVLPG instruction, thereby emulating the behavior of the physical TLB. Performance may be improved by an alternative embodiment, which may reduce the number of page faults that need to be handled by the VMM. This alternative embodiment (referred to as an eager filling of the virtual TLB technique) provides a method for re-evaluating the entire guest translation data structure and the entire content of the active translation data structure(s) in response to receiving control over any operation that may require modification of the virtual TLB.

Figures 10A and B are flow diagrams of one embodiment of a method 1000 for responding to an operation that may require modification of a virtual TLB using an eager filling of the virtual TLB technique. Upon receiving control of such an operation, the VMM locates the guest PD using the guest OS's value for CR3 and evaluates each guest PDE.

Method 1000 begins with examining the first entry in the guest PD (processing block 1003) and making a determination as to whether this guest PDE is marked not present (i.e., P=0) or not accessed (A=0) (decision box 1004). If the determination is positive, the active user PDE and active supervisor PDE are marked not present (processing block 1006), and method 1000 proceeds to decision box 1058. Alternatively, a further determination is made as to whether a physical address contained in the guest PDE is valid for the virtual machine being supported (decision box 1008). If the address is invalid, the active user PDE and active supervisor PDE are marked not present (processing block 1006), and method 1000 proceeds to decision box 1058.

If the address is valid for this VM, at processing block 1012, the PS flag in the active user PDE and the active supervisor PDE is set to match the value of this flag in the guest PDE. At processing block 1014, the U/S flag in the active supervisor PDE is set to 1 and the U/S flag in the active user PDE is set to the value of this flag in the guest PDE.

Further, at decision box 1016, a determination is made as to whether the guest PDE is for a 4 MB page (PS=1) and is marked not dirty (D=0) (decision box 1016). If this determination is positive, the R/W flag is set to 0 in both the active user PDE and the active supervisor PDE (processing block 1018), and method 1000 proceeds to decision box 1058. Otherwise, if the guest PDE is for a page table or marked dirty, then the R/W flag in the active user PDE is set to this flag's value in the guest PDE (processing block 1020) and a further determination is made as to whether the guest software maintains CR0.WP=0 (decision box 1022). If the guest software does expect CR0.WP to be equal to 0, then the R/W flag in the active supervisor PDE is set to 1 (processing block 1026). Alternatively, if the guest software expects CR0.WP to be set to 1, then the R/W flag in the active supervisor PDE is set to the value of this flag in the guest PDE (processing block 1024).

Next, each PTE in the guest page table that is referred to in the guest PDE is evaluated. In particular, the evaluation begins with the first entry in the guest PT (processing block 1028) and, at decision box 1030, a determination is made as to whether this guest PTE is marked not present or not accessed. If the determination is positive, then both the active user PTE and the active supervisor PTE are marked not present, and method 1000 proceeds to decision box 1054. Alternatively, a further determination is made as to whether a physical address contained in the guest PTE is valid for the virtual machine being supported (decision box 1034). If the address is invalid, the active user PTE and active supervisor PTE are marked not present (processing block 1032), and method 1000 proceeds to decision box 1054.

If the address is valid for this VM, at processing block 1038, the PS flag in the active user PTE and the active supervisor PTE is set 0. At processing block 1014, the U/S flag in the active supervisor PTE is set to 1 and the U/S flag in the active user PTE is set to the value of this flag in the guest PTE.

Further, at decision box 1042, a determination is made as to whether the guest PTE is marked not dirty (D=0) (decision box 1042). If this determination is positive, the R/W flag is set to 0 in both the active user PTE and the active supervisor PTE (processing block 1044), and method 1000 proceeds to decision box 1054. Otherwise, if the guest PTE is marked dirty, then the R/W flag in the active user PTE is set to this flag's value in the guest PTE (processing block 1046) and a further determination is made as to whether the guest software maintains CR0.WP=0 (decision box 1048). If the guest software does expect CR0.WP to be equal to 0, then the R/W flag in the active supervisor PTE is set to 1 (processing block 1050). Alternatively, if the guest software expects CR0.WP to be set to 1, then the R/W flag in the active supervisor PTE is set to the value of this flag in the guest PTE (processing block 1052).

Next, at decision box 1054, a determination is made as to whether more entries remain in the guest PT. If the determination is positive, method 1000 moves to the next guest PTE and its examination begins at decision box 1030. After all the entries in the guest PT are examined, the examination of other guest PDEs continues until no more entries remain in the guest PD.

Figure 11 is a block diagram of one embodiment of a processing system. Processing system 1100 includes processor 1120 and memory 1130. Processor 1120 can be any type of processor capable of executing software, such as a microprocessor, digital signal processor, microcontroller, or the like. Processing system 1100 can be a personal computer (PC), mainframe, handheld device, portable computer, set-top box, or any other system that includes software.

Memory 1130 can be a hard disk, a floppy disk, random access memory (RAM), read only memory (ROM), flash memory, or any other type of machine medium readable by processor 1120. Memory 1130 can store instructions for performing the execution of the various method embodiments of the present invention such as methods 400, 500, 700, 880, 950 and 1000 (**Figures 4, 5, 7A-7D, 8, 9, 10A and 10B**).

It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reading and understanding the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.